2

NASA Contractor Report 182100
ICASE Report No. 90-59

ICASE

MULTIPROCESSORS AND RUNTIME COMPILATION

Joel Saltz
Harry Berryman
Janet Wu

Contract No. NAS1-18605
September 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

MULTIPROCESSORS AND RUNTIME COMPILATION[1]

Joel Saltz
Harry Berryman
Janet Wu

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665

ABSTRACT

Runtime time preprocessing plays a major role in many efficient algorithms in computer science, as well as playing an important role in exploiting multiprocessor architectures. We give examples that elucidate the importance of run time preprocessing and show how these optimizations can be integrated into compilers. To support our arguments, we describe transformations implemented in prototype multiprocessor compilers and present benchmarks from the iPSC2/860, the CM-2, and the Encore Multimax/320.

[illegible]

[1] Research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665. This work was also supported under NSF grant ASC-8819374.

1 Introduction

1.1 Overview

In many algorithms, data produced or input during a program's initialization plays a large role in determining the nature of the subsequent computation. When the data structures that define a computation have been initialized, a preprocessing phase follows. Vital elements of the strategy used by the rest of the algorithm are determined by this preprocessing phase. To effectively exploit many multiprocessor architectures, we may also have to carry out run time preprocessing. This preprocessing will be referred to as *runtime compilation.* The purpose of runtime compilation is not to determine which computations are to be performed but instead to determine how a multiprocessor machine will schedule the algorithm's work, how to map the data structures and how data movement within the multiprocessor is to be scheduled. In this paper, we specifically address problems for which computational patterns can be predicted when values assigned to key data structures are known. These problems include computations on non-uniform meshes, sparse direct factorization which does not involve pivoting and sparse iterative linear solvers. (KC)

Values obtained during program execution can affect the nature and degree of potential concurrency. *Runtime compilation may be needed to identify and exploit concurrency.* Complex heterogeneous memory hierarchies characterize virtually all multiprocessor architectures with more than a few dozen processors. Primary memory is divided among processors. To obtain data from other portions of the primary memory of the multiprocessor, we typically need to access a communications network. *Program performance can be dramatically affected by the scheduling of data movement among processors.*

There has been much research carried out on methods for runtime parallelization as well as runtime workload and data partitioning. Most parallelization and problem partitioning methods explicitly or implicitly specify patterns of interprocessor communication. When patterns of computation are determined by data structures initialized during program execution, traditional compiler techniques cannot possibly carry out these partitioning and scheduling operations. Only recently have methods been developed that can integrate the kinds of runtime optimizations mentioned above into compilers and programming environments.

2 Algorithmic Execution Time Preprocessing

In many efficient approaches to solving problems in computing, data produced or input during program execution plays a large role in determining computational patterns. Examples include:

- Most searching and sorting problems
- Critical path analysis
- Game tree and decision tree manipulations
- Direct and iterative sparse linear system solvers

Once an appropriate subset of the input (or generated) data is available, it is frequently worthwhile to perform some preprocessing. This preprocessing can take many forms, but results of the preprocessing determine vital elements of the strategy used by the remainder of the algorithm. A simple example of this is the method of interpolated binary search. The number of computations required for a simple binary search of a sorted list depends on the values of the elements in the list and on the value of the key. We can preprocess the sorted list and use the distribution of element values in the list to produce an interpolation function that is used to direct the search. It is frequently possible to amortize the cost of preprocessing. In the interpolated binary search example, once preprocessing is carried out, we can used the interpolation function to search for a sequence of different keys.

Some other examples of well known algorithms that carry out preprocessing to determine vital elements of the strategy used by the remainder of the algorithm are:

a. Creation of indices to speed database retrieval where indices are created to allow the use of efficient search methods on many different database keys [30].

b. Generation of threaded binary search trees where extra links are added to a binary tree to speed tree traversal [16].

c. Matrix reordering and symbolic factorization used in sparse direct linear equation solvers. In such problems, the number and pattern of computations in a sparse matrix factorization is determined by the order in which steps in the factorization are carried out. In many cases it is possible to use the non-zero structure of a matrix to predetermine the order in which computations will be carried out and to allocate the memory needed to store the resulting factored matrix [12].

In each of these examples, the results of a single preprocessing computation can be used to solve any member of a class of structurally similar problems. In the database example, the creation of an index can be followed by an arbitrary number of queries. Once a threaded binary search tree is generated, the resulting data structure can be used in an arbitrary number of tree traversals. A symbolic matrix factorization can be used to speed the factorization of any matrix with a given pattern of non-zero entries.

Runtime compilation techniques attempt to discover how to maximize the performance of algorithms on multiprocessors. Since these methods are particularly useful in algorithms whose computational patterns depend on values assigned to data structures during program execution, a significant preprocessing cost is frequently involved. In runtime compilation, we are also often able to amortize costs of preprocessing among a number of structurally similar computational phases.

3 Run-Time Parallelization

Run-time parallelization is perhaps the most obvious form of multiprocessor runtime compilation. Parallelization carried out during compilation is necessarily conservative. If a compiler cannot figure out how to generate a correct parallelizing loop transformation, loop iterations have to be performed sequentially. Many loop nests defy compile-time parallelization because dependency patterns are determined by variables or arrays initialized during program execution. One way of carrying out runtime parallelization is to analyze the inter-iteration dependency pattern in a loop nest to identify *wavefronts* of concurrently executable loop iterations. Using a form of run time preprocessing, we transform a loop nest with inter-iteration dependencies into a sequence of parallel loops. Execution time preprocessing is frequently used to parallelize sparse numerical algorithms, such as those arising in sparse direct and iterative linear solvers [2], [4], [24], [11], [1].

Typically, programmers need to explicitly code the procedures that carry out the necessary run time preprocessing. It is possible to produce a *runtime parallelization* program transformation that generates code designed to perform run-time loop parallelization [28]. The compiler transforms a loop into *two* separate code segments. The first code segment, the *inspector*, finds sets of independent loop iterations while the second code segment, the *executor*, carries out the scheduled work. Runtime parallelization transformations have been implemented in a prototype compiler targeted at shared memory machines [29]. Runtime compilation only handles a subset of the possible types of runtime parallelization. Our transformations only apply to loop nests in which inter-iteration dependencies do not depend on the results of computations carried out within the loop nest. There are a number of algorithms that merge the process of identifying and performing concurrent work. It seems likely to us that it will be possible to produce compiler transformations that generate hybrid inspector/ executors for more fully dynamic algorithms, but we will not address this issue

```
do i=1,N

     y(i) = a(i)*y(ia(i)) + b(i)*y(ib(i))

end do
```

Figure 1: Sequential Code to be Parallelized

further in this paper.

To clarify the scheme, we now present a simple example. A simple sequential program is presented in Figure 1. Note that right hand side references to array y use a level of indirection. The inspector used to perform runtime parallelization (Figure 2) is simply a topological sort. This sort can be generated from the parse tree produced by the loop in Figure 1. The inspector in Figure 2 is sequential but can be parallelized using the principles to be described later in this section. Once the wavefront corresponding to each index is known, we can sort the indices in order of increasing wavefront number to produce the array `schedule`. The inspector also initializes a pointer array `count`. Array `count` contains the address in `schedule` of the beginning of each wavefront. Loop iterations corresponding to wavefront i are found in `schedule` between `count(i)` and `count(i+1)-1`.

The executor in Figure 3 is a sequence of parallel do loops that run over consecutive wavefronts obtained by the inspector from the sequential code in Figure 1. Note that to obtain the correct solution in the executor we need to maintain *two* copies of the array y found in the sequential code. In Figure 1 we call these copies y and ynew.

In evaluating the usefulness of run-time parallelization, the cost of the preprocessing must be taken into account. In [29] we present timings obtained from the run-time parallelization transformation applied to sparse lower triangular solves. On an 18 processor Encore Multimax/320, a single processor required 241 milliseconds to solve a lower triangular system obtained from an incomplete factorization of one of the Boeing Harwell test matrices. On 16 processors of the Multimax, the inspector required 100 milliseconds and the executor required 23 milliseconds. In many situations, we can amortize the cost of an inspector because we need to repeatedly carry out a given pattern of computations. For instance, in iterative linear systems solvers we may need to repeatedly solve the same sparse triangular systems with different right hand sides.

A variety of tradeoffs can be made between the costs and benefits of preprocessing. We

```
wf(1:N) = 0

do i=1,N

      wf(i) = max(wf(i),wf(ia(i)),wf(ib(i))) + 1

end do

Use wf() to produce schedule(), a list of indices in order of increasing wavefront number
```

Figure 2: Parallelizing Inspector

```
do phase = 1, np

      parallel do i=count(phase),count(phase+1)-1
           ii = schedule(i)

           if(ia(ii).lt.ii) then tmp1 = ynew(ia(ii))
           else tmp1 = y(ia(ii)) endif

           if(ib(ii).lt.ii) then tmp2 = ynew(ib(ii))
           else tmp2 = y(ib(ii)) endif

           ynew(ii) = a(ii)*tmp1 + b(ii)*tmp2
      end parallel do

end do

y(1:n) = ynew(1:n)
```

Figure 3: Parallelizing Executor

can dispense with reordering loop iterations into concurrent wavefronts and still be able to exploit parallelism to a degree by using a *preprocessed doacross* transformation [27]. In a doacross construct [9], loop iterations are partitioned between processors in a striped fashion and synchronization calls are introduced so that computations from some loop iterations can be overlapped. Doacross loops typically make use of a-priori knowledge of inter-iteration dependencies to carry out needed inter-iteration synchronizations. It is possible to carry out a relatively small amount of run time preprocessing and postprocessing that eliminates the need for a-priori knowledge of dependencies. On machines with snooping caches (such as the Multimax/320), it is efficient to synchronize using shared arrays. The following is a sketch of some of the transformations involved in generating preprocessed doacross loops, a much more detailed description may be found in [27]. A shared array `ready` is initialized to `NOTDONE`. When a left hand side array element `i` is calculated, `ready(i)` is set to `DONE`. Processors needing to use an updated value of array element `i` busy wait on `ready(i)` until `ready(i)` is set to `DONE`. In *preprocessed doacross loops*, we need to maintain two copies of shared arrays that appear on the left hand side of expressions during the computation. After the loop is completed, the two shared array copies need to be reconciled.

The run time initialization and postprocessing in the preprocessed doacross loop are relatively inexpensive compared to the preprocessing costs incurred by a parallelizing inspector (e.g. figure 2). For the above cited lower triangular solve involving the incompletely factored Boeing Harwell test matrix, the preprocessed doacross loop requires 45 milliseconds. This can be compared to the 23 milliseconds required to carry out the runtime parallelized solve and the 100 millisecond preprocessing time of the inspector.

Runtime parallelization can be carried out on a variety of architectures. In this paper, we discuss runtime parallelization only in the context of shared memory architectures; a discussion of runtime parallelization for distributed memory machines is found in [26].

4 Runtime Compilation for Distributed Memory Machines

4.1 Distributed Memory Inspectors and Executors

In distributed memory machines, large data arrays need to be partitioned between local memories of processors. These partitioned data arrays are called *distributed arrays*. We follow the usual practice of assigning long term storage of distributed array data to specific memory locations in the distributed machine. A processor that needs to read an array element must fetch a copy of that element from the memory of the processor in which that array element is stored. Alternately, a processor may need to store a value in an off-processor

Each processor P:

- preprocesses its own loop iterations
- Records off-processor fetches and stores in hashed cache
- Finds send/receive calls required for data exchange
  - (i) P generates list of all off-processor data to be fetched
  - (ii) Other processors tell P which data to send
  - (iii) Send/Receive pairs generated and stored

Figure 4: Inspector For Parallel Loop on Distributed Memory Multiprocessor

distributed array element. Local copies of off-processor distributed array elements are stored in hash tables called *hashed caches.* Run-time procedures carry out the movement of data between processors and manage the above mentioned hash tables.

In distributed memory MIMD architectures, there is typically a non-trivial communications latency or startup cost [7]. For efficiency reasons, information to be transmitted should be collected into relatively large messages. The cost of fetching array elements can be reduced by precomputing what data each processor needs to send and to receive.

In Figure 4, we outline the preprocessing we performed to implement a parallel loop on a distributed machine. The distribution of parallel loop indices to processors determines where computations are to be performed. We assume that all needed distributed arrays have been defined and initialized and that loop iterations have been partitioned between processors. Using the hashed cache to record off-processor fetches and stores allows us to recognize when more than one reference is being made to the same off-processor distributed array element, so that only one copy of that element need be fetched or stored.

During our inspector phase, we carry out a set of interprocessor communications that allows us to anticipate exactly which send and receive communication calls each processor must execute so that all interprocessor data transmission is correctly carried out. By contrast, if individual fetches and stores were to be carried out during the actual computation, things would be much more awkward. For example, in such a case processor A might obtain the contents of a distributed array element which is not on A by sending a message to processor B associated with the array element. Processor B would be programmed to anticipate a request of this type, to satisfy the request and to return a responding message containing the contents of the specified array element.

- Before loop or code segment

  (i) Data to be sent off-processor read from distributed arrays

  (ii) Send/receive calls transport off-processor data

  (iii) Data written into hashed cache

- Computation carried out

  - off-processor reads/writes go to hashed cache

- At end of loop or code segment

  (i) Data to be stored off-processor read from hashed cache

  (ii) Send/receive calls transport off-processor data

  (iii) Data written back into distributed arrays for longer term storage

Figure 5: Executor For Parallel Loop on Distributed Memory Multiprocessor

Once preprocessing is completed, we are in a position to carry out the necessary communication and computation, Figure 5 outlines the steps involved. The initial data exchange phase follows the plan established by the inspector. During preprocessing, each processor finds out which distributed array elements need to be transmitted. When a processor obtains copies of off-processor distributed array elements, the copies are written into the processor's hashed cache. Once the communication phase is over, each processor carries out its computation. Each processor uses locally stored portions of distributed arrays along with off-processor distributed array elements stored in the hashed cache. When the computational phase is finished, distributed array elements to be stored off-processor are obtained from the hashed cache and sent to the appropriate off-processor locations.

There are many situations in which simple, easily specified distributed array partitions are inappropriate. For instance when we compute using an unstructured mesh, we attempt to partition the problem so that each processor performs approximately the same amount of work and so that the communications overhead is minimized. Typically, it is not possible to

Each processor P:

- preprocesses its own loop iterations
- Records off-processor fetches and stores in hashed cache
- Consults distributed translation table to
  - Find location in distributed memory for each off-processor fetch or store
- Finds send/receive calls required for data exchange
  - (i) P generates list of all off-processor data to be fetched
  - (ii) Other processors tell P which data to send
  - (iii) Send/Receive pairs generated and stored

Figure 6: Inspector For Parallel Loop Using Irregular Distributed Array Mapping

express the resulting array partitions in a simple way. If we allow an arbitrary assignment of distributed array elements to processors, the data structure used to describe the partitioning will have the same number of elements as the distributed array.

In order to access an array element, we need to know where the element is stored in the memory of the distributed machine. We use a distributed translation table defined by a partitioning algorithm, to describe the mapping. When a distributed translation table is used to describe array mappings, inspectors must be modified so that they access the distributed table. Using an irregular array mapping *does not alter the form of the executor.* The modifications to be made to an inspector are outlined in Figure 6.

4.2 Languages and Tools for Irregular Problems

Programs designed to carry out sparse direct and iterative methods also typically require many of the optimizations described in Section 4.1. Some examples of such programs are described in [3], [18], [15], [4]. Williams [34] describes a programming environment (DIME) for calculations with unstructured triangular meshes using distributed memory machines. In [34], collections of distributed array accesses are translated into an efficient set of inter-node messages. The DIME programming environment embodies many of the principles discussed in Section 4.1. The optimizations discussed in the last section can be incorporated into distributed compilers. Runtime compilation for distributed machines was proposed in [25]; this description was in the context of the Crystal language. Distributed memory runtime compilation was expanded upon in [20]; which outlines the principals behind the PARTI project. A more detailed description of the concepts behind distributed memory runtime compilation is found in [26], and [21]. The idea of splitting a loop into an inspector and executor and integrating this into a compiler was also developed independently as part of the KALI project [17]. Other compiler projects have also proposed run time resolution of communications on distributed machines [8], [22], [23]. These compilers do not carry out the kinds of run time optimizations of the sort described here.

We have designed a set of procedures or primitives that do the work needed to implement inspectors and executors. We have also designed and implemented a model compiler that recognizes a subset of Fortran (ARF - ARguably Fortran) and generates inspector and executor loops with embedded primitives. Distributed arrays can be declared in ARF source. These distributed arrays can either be partitioned between processors in a uniform manner (e.g. equal sized blocks of contiguous array elements assigned to each processor), alternately, distributed arrays can be partitioned in an irregular manner. When an array is to be partitioned in an irregular fashion, mapping information is specified in an integer array. This integer array is typically produced by a partitioning procedure. Element `i` of the integer array describes the processor to which element `i` of the distributed array is to be mapped. For example, consider the ARF declaration, `distributed irregular using map real y(4)`. This declaration denotes a four element real array y that is to be distributed according to integer array `map`.

Embedded primitives include communications procedures designed to support irregular patterns of distributed array access. Other primitives that involve interprocessor communication initialize distributed translation tables or access distributed translation tables to find the location of irregularly mapped distributed array data. Primitives also support the maintenance of hashed caches. (Recall from Section 4.1 that hashed caches store copies of off-processor distributed array data.) There are also PARTI primitives that perform accumulations to off-processor distributed array elements.

In Figure 7 we present a simple example of an ARF program. The procedure to be presented is a block sparse matrix vector multiply, obtained from an iterative solver produced for a program designed to calculate fluid flow for geometries defined by an unstructured mesh [31]. The ARF compiler uses information in integer array `row` to make calls to primitives that initialize the distributed translation tables. These distributed translation tables are used to describe the mapping of `x, y , cols` and `ncols` (statements S1 and S2). The current version of ARF distributes only the last declared dimension in a multidimensional array, although the PARTI primitives do support a broader class of array mappings [6]. As of the time of writing, the ARF compiler does not include syntax that specifies where computational work is to be performed. Partitioning procedures specify where work is to be carried out, but the interface between partitioning procedures and ARF has not yet been automated.

In Figure 7, array x is indexed by m and `cols(j,i)`. If we were to carry out the work in this loop in a naive manner, we would have to fetch each individual distributed array element `x(m,cols(j,i))` (statement S4) from its assigned processor. Since the processor assignments of elements of x are stored in a distributed translation table, we would also need to access the memory of the processor that keeps track of where `x(m,cols(j,i))` is stored.

In Table 1, we present the execution times on 32 processor and 64 processor Intel iPSC/860 machines, obtained from the block matrix vector multiply kernel as well as execution times from another, more complex kernel that arose in an unstructured code. This kernel, to be referred to here as *fluxroe*, computes convective fluxes using a method based on Roe's approximate Riemann solver [32], [33]; the kernel is discussed in some detail in [6]. Both the block matrix vector multiply and the fluxroe kernel arise from iterative algorithms. In these tests, fluxroe was translated into ARF and compiled. In these experiments, we used two different unstructured meshes:

(i) A 21,672 element mesh generated to carry out an aerodynamic simulation involving a multielement airfoil in a landing configuration [19]

(ii) A 37,741 element mesh generated to simulate a 4.2 % circular arc airfoil in a channel [14].

In all the cases presented below, each unstructured mesh was partitioned by recursive orthogonal dissection [13].

In table 1 we present:

inspector time - time required to carry out the inspector preprocessing phase

computation time - the time required to perform computations in the iterative portion of the program

communication time - the time required to exchange messages within the iterative portion of the program.

The inspector time includes the time required to set up the needed distributed translation table as well as the time required to access the distributed translation table when carrying out the preprocessing in the inspector. In these experiments, the ratio of the time required to carry out the inspector to the computation time required for a single iteration ranged from a factor of 3 to a factor of 5. Most of the preprocessing time goes to setting up and using the distributed translation table. For instance, consider the block matrix vector multiply on 64 processors using the 21,672 element mesh. The total preprocessing cost was 122 milliseconds, of which 111 milliseconds went to translation table related work.

We can define parallel efficiency for a given number of processors P as the sequential time divided by the product of the execution time on P processors times P. In Table 1 we depict under the heading of *single sweep efficiency*, the parallel efficiencies we would obtain were we required to preprocess the kernel each time we carried out calculations. In reality, preprocessing time can be amortized over multiple mesh sweeps. If we neglect the time required to preprocess the problem in computing parallel efficiencies, we obtain the second set of parallel efficiency measurements, the *amortized efficiency* presented in Table 1. The amortized efficiencies for 64 processors ranged from 0.48 to 0.59, while the single sweep efficiencies ranged from 0.10 to 0.17.

In the experiments depicted in Table 1, the time spent computing is at least a factor of 2 greater than the communication time. The amortized efficiencies are, however, impacted by the fact that the computations in the parallelized codes are carried out less efficiently than those in the sequential program. The parallel code spends time accessing the hashed cache. It also needs to perform more indirections than the sequential program.

Table 1: Performance on different number of processors

nprocs nprocs	inspector time(ms)	comp time(ms)	comm time(ms)	single sweep efficiency	amortized efficiency
Block Matrix Vector Multiply - 21,672 element mesh					
32	148	49	9	0.15	0.55
64	122	25	9	0.10	0.48
Block Matrix Vector Multiply - 37,741 element mesh					
32	200	85	10	0.19	0.59
64	150	42	9	0.14	0.54
Fluxroe - 21,672 element mesh					
8	231	310	24	0.40	0.69
16	162	157	21	0.34	0.65
32	135	80	22	0.19	0.57
64	172	41	19	0.12	0.48
Fluxroe - 37,741 element mesh					
8	393	534	23	0.41	0.70
16	249	269	18	0.36	0.68
32	191	156	23	0.28	0.62
64	203	69	14	0.17	0.59

```
S1 distributed irregular using row real x(4,n), y(4,n)

S2 distributed irregular using row integer cols(9,n), ncols(n)

   ... initialization of local variables ...

   doall i=1,n

        do j=1,ncols(i)

         S3 do k=1,4

                sum = 0

                do m = 1,4

            S4  sum = sum + f(i,m,k,i)*x(m,cols(j,i))

                enddo

            y(k,i) = y(k,i) + sum

            enddo

        enddo

   enddo
```

Figure 7: ARF Kernel From Unstructured CFD Code

4.3 Future Optimizations

Most of the optimizations described in this section are motivated either directly or indirectly by the high communication latencies typically found in distributed memory computers. Because we can anticipate all of the interprocessor communications that will be needed in carrying out a loop, we have the information we need to schedule interprocessor communications to reduce overheads due to contention. As we shall see in Section 5, scheduling of interprocessor communication has already been shown to be an important optimization for some SIMD architectures. We expect this to also turn out to be a fruitful optimization for distributed memory MIMD computers.

Computations can be characterized by patterns of data dependency. Procedures that partition data structures and computational work take these dependency patterns into account. It is possible to design program transformations that generate procedures which output a record of the dependency patterns in a loop nest in a a standard representation [20]. Standardized partitioning programs that use these data structures can then be employed.

5 Runtime Compilation in SIMD Machines - the Communications Compiler

Irregular problems can cause serious performance degradation on the CM-2 [5]. It turns out that this performance degradation can be ameliorated by a form of runtime compilation. Denning Dahl has developed a set of software facilities for the Connection Machine (CM-2) that are designed to handle applications that exhibit fixed irregular patterns of communication [10]. One procedure, the *communications compiler* schedules interprocessor communications. The other procedure a *mapping facility* maps graphs generated from a communication pattern onto the CM-2. In this paper, we will focus our attention on the communications compiler.

The communications compiler decomposes an irregular communications pattern into a sequence of simple, inexpensive data transfers. These data transfers make use of the hypercube communication network in the CM-2. In the CM-2, all links of the hypercube can simultaneously carry bidirectional information. The communications compiler attempts to reduce time required for communication by the judicious scheduling of messages. At present, the communications compiler is accessed by procedure calls. Lists of destination addresses are passed to the communication compiler's preprocessor procedures. Once the preprocessing is completed, a data delivery function carries out the scheduled communications. Transformations analogous to those discussed in Section 4.2 could be used to embed these communication compiler primitives into programs, and hence to generate inspectors and executors.

We present a set of benchmarks that quantifies the performance effects of the communications compiler. A synthetic workload was defined in the following way. A square mesh in which each point was linked to four nearest neighbors was incrementally distorted. Random edges were introduced subject to the constraint that in the new mesh, each point still required information from four other mesh points.

The following assumptions are inherent in our workload generator: makes

(i) The problem domain consists of a 64 by 128 mesh of points which are numbered using their row major or natural ordering;

(ii) Each point is initially connected to its four nearest neighbors

(iii) Each link produced in the above step is examined, with probability q the link is replaced by a link to a randomly chosen point.

An 8192 processor Connection Machine-2 was configured as a 64 by 128 torus. The mesh was mapped onto the torus in the obvious manner. A sweep over the mesh was then performed using the following communication mechanisms:

(i) Get: The standard CM-2 general router is called four times, once for each of the four off-processor data elements needed by each processor.

(ii) Compiled get: Communications compiled using the communications compiler; the communications compiler preprocessor was called four times, once for each of the four off-processor data elements required by each processor. The data delivery procedure is called four times during each mesh sweep.

(iii) Compiled gather: Communications compiled using the communications compiler; a single call to the communications compiler preprocessor handles each processor's four data requests. For each iteration, a single data delivery function carries out all communication.

(iv) NEWS: CM-2 communications procedures that transmit information using mesh embedded into hypercube by binary reflected gray code. NEWS was only used to benchmark the completely uniform mesh (q = 0).

The construction of the communication schedule took anywhere from 1 to 13 *seconds*.

The results of these benchmarks are depicted in Figure 8. In these experiments we carried out sweeps over meshes generated by varying q from 0.0 to 0.5. For the uniform mesh (q=0), we used all four communications mechanisms described above. For the synthetically generated irregular meshes, we used the standard CM get,the compiled get and the compiled gather. Let T_{NEWS} represent the the time required by the CM-2 to to sweep over a regular mesh (q=0) using the NEWS mechanism; T_{NEWS} was equal to 0.80 milliseconds. In Figure 8, we compare T_{NEWS} with the time taken by the CM-2 to sweep over irregular meshes using the standard CM get (T_{GET}), the compiled get (T_{CGET}) and the compiled gather ($T_{CGATHER}$). For the regular mesh, T_{GET}, T_{CGET}, and $T_{CGATHER}$ were factors of 15.4, 2.2 and 1.1 times larger than T_{NEWS}. As q increased, the performance of the mesh sweep degraded significantly with all three routing mechanisms tested. For $q = 0.5$, T_{GET}, T_{CGET}, and $T_{CGATHER}$ were factors of 22.6, 4.4 and 2.7 times larger than T_{NEWS}. It is clear that runtime compilation techniques can play an important role in reducing communications costs for irregular problems on SIMD machines. The computational cost of the simulated annealing based communications compiler is, however, extremely high.

6 Conclusions

Execution time preprocessing plays a major role in many efficient algorithms in computer science. Runtime preprocessing also plays an important role in exploiting multiprocessor architectures. Examples of such preprocessing include runtime parallelization, runtime aggregation and scheduling of remote distributed array accesses and execution time data and workload partitioning. We have given examples of how optimizations of this type can be integrated into compilers. We have also presented specific benchmarks that document, on a range of multiprocessor architectures, the importance of various types of runtime compilation.

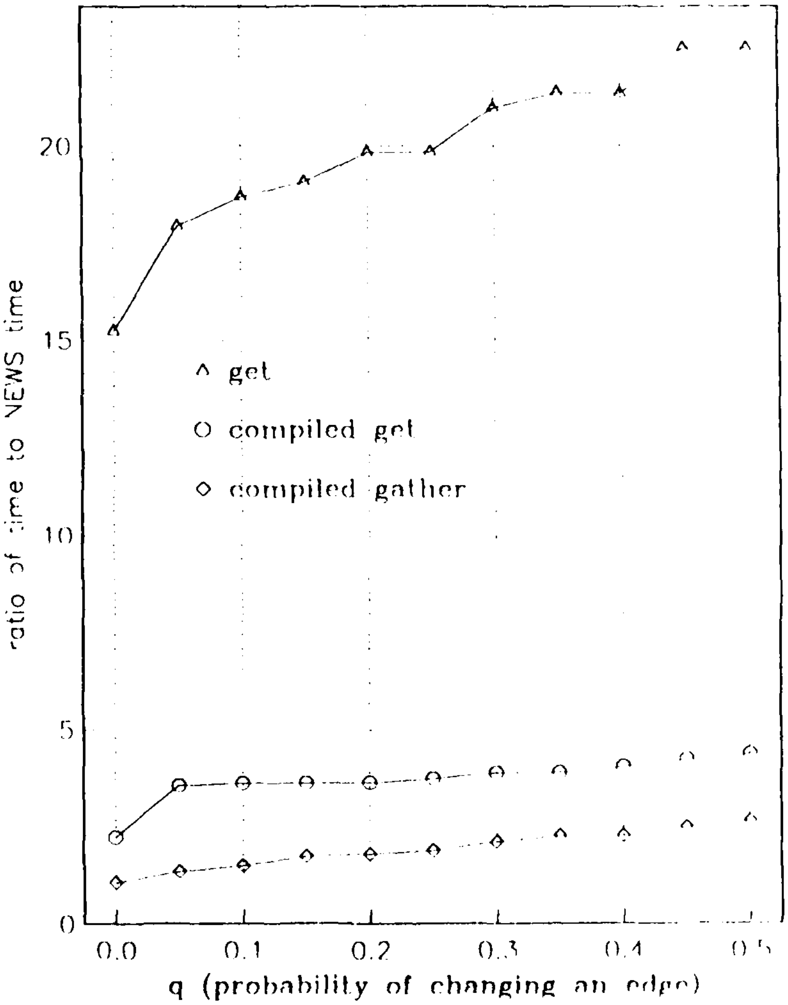

Figure 8: Effect of Communications Compiler - Synthetic Workload

7 Acknowledgements

We wish to thank Dimitri Mavriplis and David Whitaker for supplying us with unstructured meshes, and to David Whitaker and P Venkatkrishnan for allowing us to use computational kernels from their codes. We would also like to thank Adam Rifkin for helping to code the partitioning algorithms and for carrying out iPSC/860 timings. We would like to thank the Advanced Computing Laboratory at at Oak Ridge National Lab for allowing us to use their 128 node iPSC/860 and Ravi Mirchandaney and Bob Voigt for their helpful comments on the manuscript.

References

[1] G. Alaghband, *Parallel pivoting combined with parallel reduction*, Parallel Computing Journal, (1989), pp. 201–221.

[2] E. Anderson and Y. Saad, *Solving sparse triangular linear systems on parallel computers*, International Journal of High Speed Computing, 1 (1989), pp. 73–95.

[3] C. Ashcraft, S. C. Eisenstat, and J. W. H. Liu, *A fan-in algorithm for distributed sparse numerical factorization*, SISSC, 11 (1990), pp. 593–599.

[4] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley, *An experimental study of methods for parallel preconditioned krylov methods*, in Proceedings of the 1988 Hypercube Multiprocessor Conference, Pasadena CA, January 1988, pp. 1698,1711.

[5] H. Berryman, J. Saltz, and W. Gropp, *Krylov methods with incomplete factorization preconditioners on the cm-2*, Journal of Parallel and Distributed Computing, 8 (1990), pp. 186–190.

[6] H. Berryman, J. Saltz, and J. Scroggs, *Execution time support for adaptive scientific algorithms on distributed memory machines*, ICASE Report 90-41, May 1990.

[7] S. Bokhari, *Communication overhead on the intel ipsc-860 hypercube*, ICASE Interim Report 90-10 , 1990.

[8] D. Callahan and K. Kennedy, *Compiling programs for distributed-memory multiprocessors*, Journal of Supercomputing, 2 (1988), pp. 151–169.

[9] R. Cytron, *Doacross: Beyond vectorization for multiprocessors*, in The Proceedings of the ICPP, 1986, 1986, pp. 836–844.

[10] E. D. Dahl, *Mapping and compiled communication on the connection machine system*, in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston S.C., 1990.

[11] I. S. Duff, *Parallel implementation of multifrontal schemes*, Parallel Computing, 3 (1986), pp. 193–204.

[12] I. S. Duff and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford Science Publications, Oxford University Press, New York, 1986.

[13] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Computers*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[14] *Numercial methods for the computation of inviscid transonic flows with shock waves - a gamm workshop*, in Notes on Numercial Fluid Mechanics, vol. 3.

[15] A. George, M. T. Heath, J. Liu, and E. Ng, *Sparse cholesky factorization on a local memory multiprocessor*, SISSC, 327-340 (1988).

[16] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville Maryland, 1976.

[17] C. Koelbel, P. Mehrotra, and J. V. Rosendale, *Supporting shared data structures on distributed memory architectures*, in 2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming, ACM SIGPLAN, Mar. 1990, pp. 177–186.

[18] J. W. Liu, *Computational models and task scheduling for parallel sparse cholesky factorization*, Parallel Computing, 3 (1986), pp. 327–342.

[19] D. J. Mavriplis, *Multigrid solution of the two-dimensional Euler equations on unstructured triangular meshes*, AIAA Journal, 26 (1988), pp. 824–831.

[20] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley, *Principles of runtime support for parallel processors*, in Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France, July 1988, pp. 140–152.

[21] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman, *A scheme for supporting automatic data migration on multicomputers*, in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston S.C., 1990.

[22] A. Rogers and K. Pingali, *Process decomposition through locality of reference*, in Conference on Programming Language Design and Implementation, ACM SIGPLAN, June 1989, pp. 69–80.

[23] M. Rosing and R. Schnabel, *An overview of Dino - a new language for numerical computation on distributed memory multiprocessors*, Tech. Rep. CU-CS-385-88, University of Colorado, Boulder, 1988.

[24] J. Saltz, *Aggregation methods for solving sparse triangular systems on multiprocessors*, SIAM J. Sci. and Stat. Computation., 11 (1990), pp. 123–144.

[25] J. Saltz and M. Chen, *Automated problem mapping: the crystal runtime system*, in The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN, September 1986.

[26] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, *Run-time scheduling and execution of loops on message passing machines*, Journal of Parallel and Distributed Computing, 8 (1990), pp. 303–312.

[27] J. Saltz and R. Mirchandaney, *The preprocessed doacross loop*, ICASE Interim Report 90-11, 1990.

[28] J. Saltz, R. Mirchandaney, and K. Crowley, *Runtime parallelization and scheduling of loops*, in Proceedings of the 1st Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, 1989.

[29] ——, *Run-time parallelization and scheduling of loops*, ICASE Report 90-34, May 1990.

[30] J. D. Ullman, *Principals of Database Systems*, Computer Science Press, Rockville Maryland, 1982.

[31] P. Venkatkrishnan, J. Saltz, and D. Mavriplis, *Parallel preconditioned iterative methods for the compressible navier stokes equations*, in 12th International Conference on Numerical Methods in Fluid Dynamics, Oxford, England, July 1990.

[32] D. L. Whitaker and B. Grossman, *Two-dimensional euler computations on a triangular mesh using an upwind, finite-volume scheme*, in Proceedings AIAA 27th Aerospace Sciences Meeting, Reno, Nevada, January 1989.

[33] D. L. Whitaker, D. C. Slack, and R. W. Walters, *Solution algorithms for the two-dimensional euler equations on unstructured meshes*, in Proceedings AIAA 28th Aerospace Sciences Meeting, Reno, Nevada, January 1990.

[34] R. D. Williams and R. Glowinski, *Distributed irregular finite elements*, Tech. Rep. C3P 715, Caltech Concurrent Computation Program, February 1989.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.
NASA CR-182100
ICASE Report No. 90-59

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle

MULTIPROCESSORS AND RUNTIME COMPILATION

5. Report Date

September 1990

6. Performing Organization Code

7. Author(s)

Joel Saltz
Harry Berryman
Janet Wu

8. Performing Organization Report No.

90-59

10. Work Unit No.

505-90-21-01

9. Performing Organization Name and Address
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

11. Contract or Grant No.

NAS1-18605

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

13. Type of Report and Period Covered

Contractor Report

14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor:
Richard W. Barnwell

Submitted to Concurrency, Practice
and Experience

Final Report

16. Abstract

Runtime time preprocessing plays a major role in many efficient algorithms in computer science, as well as playing an important role in exploiting multiprocessor architectures. We give examples that elucidate the importance of run time preprocessing and show how these optimizations can be integrated into compilers. To support our arguments, we describe transformations implemented in prototype multiprocessor compilers and present benchmarks from the iPSC2/860, the CM-2, and the Encore Multimax/320.

17. Key Words (Suggested by Author(s))

runtime compilation, shared memory,
distributed memory, massively paralleled,
inspector, executor

18. Distribution Statement

59 - Mathematical and Computer Sciences
(General)
61 - Computer Programming and Software
Unclassified - Unlimited

19. Security Classif. (of this report)
Unclassified

20. Security Classif. (of this page)
Unclassified

21. No. of pages
23

22. Price
A03

NASA-Langley, 1990